Carnegie Mellon
Software Engineering Institute

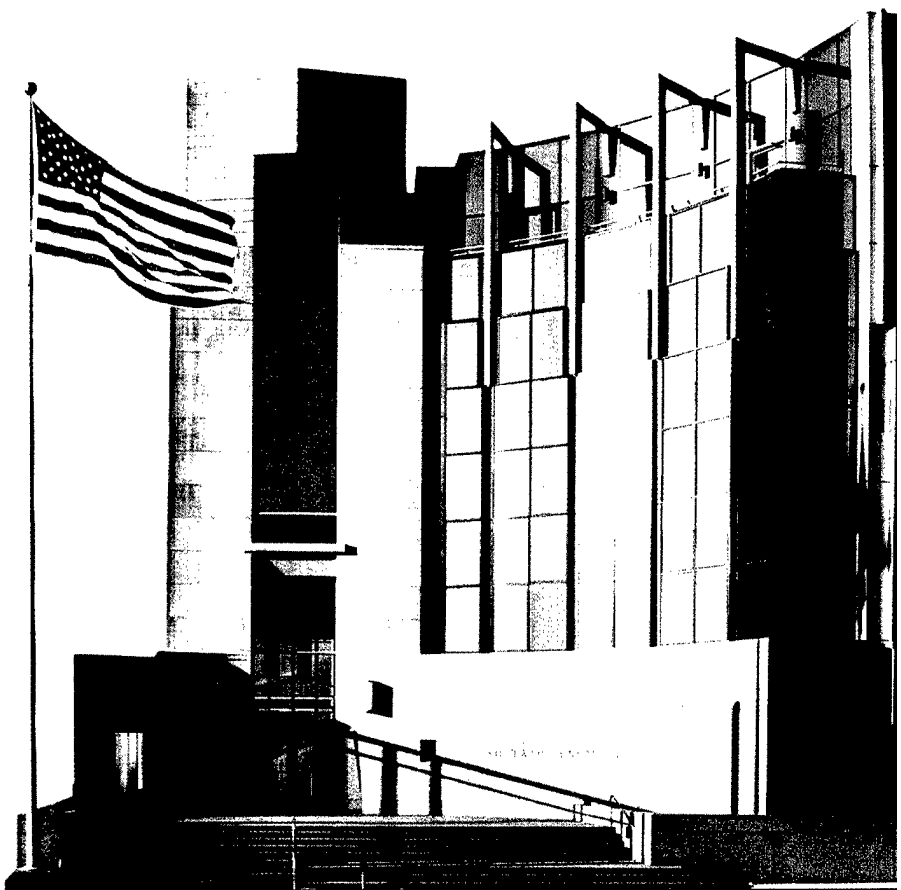
Architecture Reconstruction of J2EE Applications: Generating Views from the Module Viewtype

Liam O' Brien
Vorachat Tamarree

November 2003

Architecture Tradeoff Analysis Initiative

Technical Note
CMU/SEI-2003-TN-028



Contents

List of Figures

List of Tables

Abstract

This report outlines the application of architecture reconstruction techniques to the Sun Microsystems' Duke's Bank system—a Java2 Platform, Enterprise Edition/Enterprise JavaBeans (J2EE/EJB) application implemented mainly in Java. The goal of the reconstruction was to apply architecture reconstruction techniques to a system implemented in Java to produce a set of views that depict that system's architecture. Decomposition style views of the module viewtype were used. They focus on the "is part of" relation and show how the system is decomposed into modules and submodules.

During the reconstruction, several decomposition style views of the architecture were generated using the Understand for Java tool. That tool extracted and then abstracted low-level source information from the system. Then that information was formatted using Perl scripts, so it could be loaded into the Architecture Reconstruction and Mining (ARMIN) tool developed by the Carnegie Mellon® Software Engineering Institute and the Robert Bosch Corporation. The resulting views showed the architectural elements of the Duke's Bank system and the dependencies among them.

1 Introduction

Previously, we applied architecture reconstruction to systems in the embedded automotive domain that were implemented in C [O'Brien 01] and on a visualization system implemented in C++ [O'Brien 03]. This report outlines an architecture reconstruction carried out on the Duke's Bank system—an online banking application that is part of the Sun Microsystems tutorial on the Java2 Platform, Enterprise Edition (J2EE) [Sun 03]. We chose the Duke's Bank system because it's implemented mainly in Java, and applying reconstruction techniques to systems implemented in Java is the main focus of this reconstruction study.

The Duke's Bank system has two clients: a J2EE application client used by administrators to manage customers and accounts, and a Web client used by customers to access account histories and perform transactions. The clients access the customer, account, and transaction information maintained in a database through Enterprise JavaBeans (EJB).

The primary goal of the reconstruction was to generate views from the module viewtype [Clements 03], so we could understand how the static Java parts of the application were decomposed and identify dependencies among them. Reconstruction of the dynamic behavior and structure of the system were not carried out in this case study but will be the subject of a future technical report. The secondary goal of this work is to determine how the architecture of J2EE/EJB applications and applications implemented in Java can be reconstructed. We also wanted to determine the usefulness of the Understand for Java tool [STI 03] for parsing and analyzing the Java code, and to determine how we could extract the various elements and relations that were used in the reconstruction process from the Java code.

The reconstruction process, shown in Figure 1, consisted of the following steps:

1. Source Information Extraction: In this step, a set of elements and relations is extracted from the system and loaded into the Architecture Reconstruction and Mining (ARMIN) tool developed by the Carnegie Mellon® Software Engineering Institute and the Robert Bosch Corporation.
2. Architectural View Composition: In this step, views of the system's architecture are generated by abstracting the source information through aggregation and manipulation using ARMIN [O'Brien 03]. The views are then presented to the reconstructor who can navigate through and manipulate them.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office.

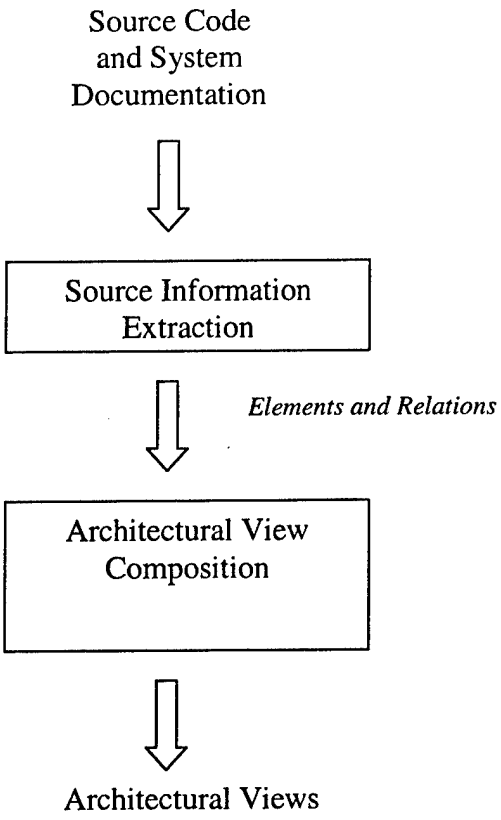


Figure 1: Reconstruction Process

The source code and any system documentation are input to the reconstruction process. Typically during that process, the system's maintainers and developers provide information about the system that helps the reconstructor generate the architectural views.[1] However in this case, because we were interested only in the static decomposition of the Java code and the dependencies among the various architectural elements, their help wasn't necessary.

We got the information we needed by parsing and analyzing the Java code, and then we used that information to reconstruct views of the architecture. In a J2EE application, the system's structure and architecture may be different at runtime than they are while the system is static.

The end result of the reconstruction process was a set of architectural views of the Duke's Bank system. These views were from the module viewtype, and showed the static decomposition of the Java code within the system and the dependencies among the various architectural elements. Using ARMIN, the user can look at, navigate through, and manipulate these views. Also, by selecting a particular component or connector between components, the user can see information about it and can even "drill down" to see information about its subcomponents.

[1] Because those people are familiar with the J2EE technology in which the system is implemented, the reconstructor would rely heavily on their input, especially if he/she didn't know much about J2EE-implemented systems.

ARMIN's Aggregator component contains an Interpreter that provides the capability of loading and running command scripts to carry out most of the tasks of Step 2 automatically. After a command script is written in an editor and loaded into ARMIN, it can be used to manipulate the data in the database and produce new views.

The remainder of this technical note is organized as follows. Section 2 describes Step 1: Source Information Extraction. Section 3 describes the reconstruction activities that are part of Step 2: Architectural View Composition, and Section 4 provides conclusions and information on future work.

2 Source Information Extraction

Before beginning Step 1: Source Information Extraction, we have to determine which information needs to be extracted from the source code. In this case, we want to understand the static structure of the J2EE/EJB application so we can document the static relationship among the architectural elements in the system. To do this, we determined which architectural styles or viewtypes [Clements 03] were appropriate. In this case, we chose to reconstruct the decomposition style from the module viewtype, because it shows how the system's responsibilities are partitioned across modules and how those modules are decomposed into submodules.

The decomposition style of architecture emphasizes the static behavior of a system. The main relation outlined in this style is "is part of." To reconstruct views of this style from the system, we identified which elements (files, classes, variables, etc.) and relations (file includes file, class has_subclass class, etc.) we needed to extract from the system to generate the decomposition style views. The element types and relations that we identified are shown in Table 1.

Relation Name	Source Element	Target Element	Explanation
defines_fn	Class	Function	A class defines a function.
contains	File	Function	A file contains a function.
defines	File	Class	A file defines a class.
defines_class	Package	Class	A package defines a class.
defines_global	File	Global_variable	A file defines a global variable.
defines_var	Function	Local_variable	A function defines a local variable.
depends_on	File	File	A file depends on another file.
has_member	Class	Member_variable	A class has a member variable.

Table 1: Identified Element Types and the Relations Among Them

We used the Understand for Java tool [STI 03] to parse and analyze the source code of the Duke's Bank system. We generated a set of textual report files that show information such as a call tree and a data dictionary for the system. To obtain the instances of the elements and relations from these report files, the tool analyzed each file to identify which source elements and relations could be extracted and used to generate decomposition style views of the architecture. Table 2 shows the types of report files generated by the Understand for Java tool

that can be used to create decomposition style views, and the relations and element types referenced within them.[2]

To ensure that information is not lost when instances of the same element types and relations are extracted from multiple files, the list of element types and relations required for decomposition style views are used as the reference for analyzing and selecting the list of report files. Table 3 shows the types of report files from which the instances of element types and relations required for this reconstruction case study were extracted.

Report Type	**Filename & Type**	**Elements**	**Relations**
Data Dictionary	bank.dic	class, type, variable, parameter, function, include file, location of the source code	has_member
Program Unit Cross Reference	bank.pux	function	Calls
Object Cross Reference	bank.obx	variable, argument	all defines_* relations, calls, sets
Class and Interface Type Cross Reference	bank.tyx	class	contains, calls, depends_on
Package and File Declaration Trees	bank.dct	package, class, methods	defines_class, contains
Class Extend Tree	bank.cet	class	Calls
Invocation Tree	bank.nvt	class, method	Calls
Simple Invocation Tree	bank.sit	class, method	Calls
Import	bank.imp	class	N/A
Program Unit Complexity	bank.cmx	class, method	defines_fn
Project Metrics	bank.jme	N/A	N/A
Class Metrics	bank.cme	class, method	N/A
Class OO Metrics	bank.cmo	class, method	N/A
Method Metrics	bank.pmx	class, method	N/A
File Metrics	bank.fmx	file	N/A
Unused Objects	bank.qno	file	N/A
Unused Types	bank.qnt	file, class	N/A
Unused Methods	bank.qnu	class, method	N/A

Table 2: The Understand for Java Tool Report Files and Their Related Information[3]

[2] Note that although additional relations and element types can be extracted to generate views from other viewtypes and documentation styles, they were not examined in this case study.

[3] The shaded rows in this table represent report files that are not usable in this particular architecture reconstruction.

Relations & Elements	Can Be Acquired from the Files
defines_fn, class, function	bank.cmx, bank.dct
contains, file, function	bank.pux, bank.dct
defines, file, class	bank.cmx, bank.dct
defines_class, package, class	bank.dct
defines_global, file, global_variable	bank.obx
defines_var, function, local_variable	bank.obx
depends_on, file, file	bank.tyx
has_member, class, member_variable	bank.dic

Table 3: Elements and Relations with Their Related File(s)

Once we identified which report files were usable in this reconstruction, we generated a script that parsed and formatted the information from the selected files, so it could be extracted and converted in the Rigi Standard Format (RSF) [Müller 93] and then loaded into ARMIN. We created Perl scripts for parsing each individual report file and producing the set of instances of the elements and relations in RSF. The scripts had to be developed and tested carefully in an iterative development process. Any duplicate relations and elements were eliminated, although ARMIN can handle duplicate instances of relations. The result of Step 1: Source Information Extraction is a single file that contains all the instances of the elements and relations from the Duke's Bank system that we needed to generate decomposition style views.

3 Architectural View Composition

The first step in composing architectural views is to load the RSF file containing the instances of the elements and relations into ARMIN [O'Brien 03]. ARMIN consists of three major components:

1. Navigator: used to create, manage, and organize projects that contain all the instances of the elements and relations that are stored in a database
2. Aggregator: used to visualize information loaded into ARMIN, and then to generate and manipulate the views that are produced during the reconstruction
3. Interpreter: used to execute command scripts for abstracting data and generating views. Those scripts can be created using a text editor and then loaded into the Interpreter for execution. This component is linked to the Aggregator.

Figure 2 shows the elements (nodes) and relations among them (edges) as displayed in an Aggregator window for the Duke's Bank system after they have been loaded into ARMIN. On the right side of the window, the Entities[4] and Relations checkboxes allow you to control which elements and relations appear in the view.

[4] In ARMIN, the term *entity* is used to represent an element. ARMIN can be used to represent elements other than software ones.

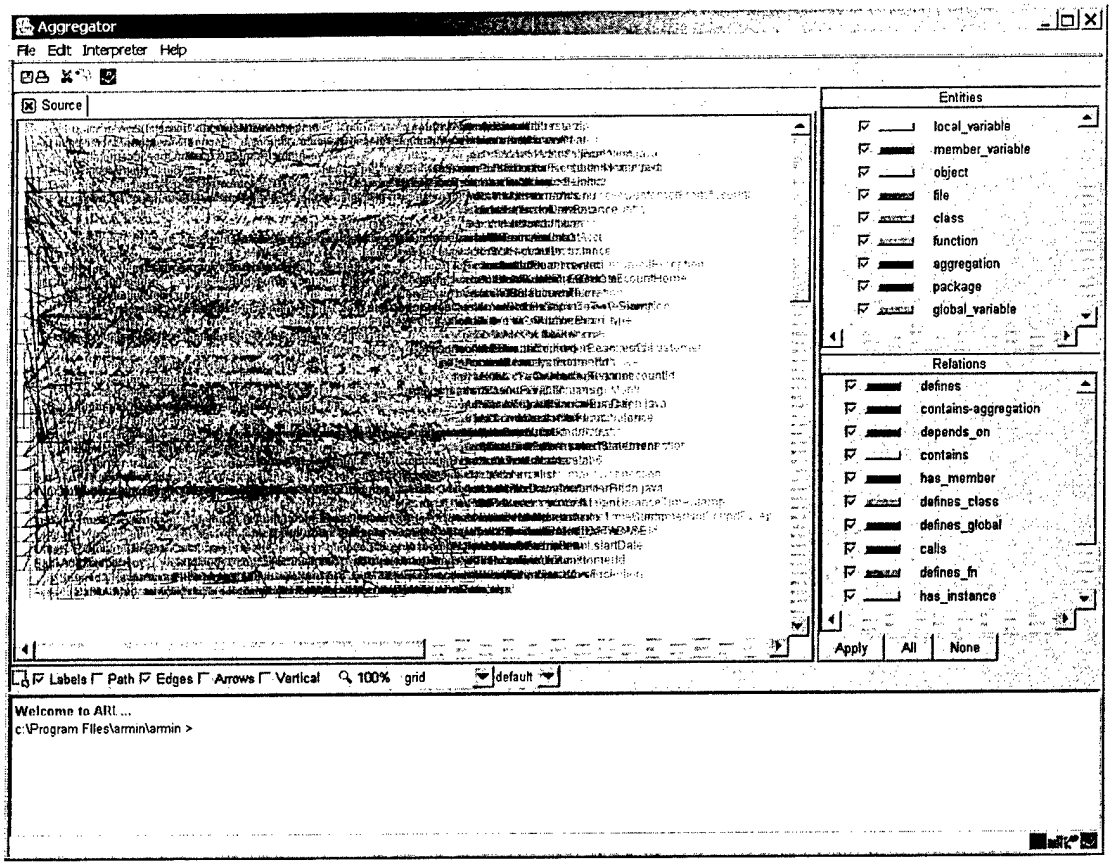


Figure 2: Sample Aggregator Window in ARMIN [5]

The view shown in Figure 2 contains all the low-level source information that was extracted from the system by the Understand for Java tool. Clearly, the graph shown in the window is unreadable and therefore unusable, so the data in it needs to be abstracted and used as input to generate higher level architectural views.

ARMIN command scripts for reconstruction are developed to aggregate elements and relations, aggregate group elements, and combine the extracted information in different ways to provide different levels of abstraction. An excerpt from such a command script is shown below.

```
#Create CLASS+ view
#collapse member functions and variables inside class
$d = desc(system.types.class);
$d.merge(/ext="+");
collapse($d,/graph="CLASS+",/type=system.types.class);
show();
```

[5] All the graphical representations generated in ARMIN are in color. The color of a box indicates the type of element, and the color of its edges indicates the type of relation.

The `desc` command in the above code generates a list of the functions and member variables within each class. Next, the `merge` command appends a plus sign (+) to the end of the class name and merges it into that list. Then, the `collapse` command removes the list of source elements (functions and member variables) from the current graph and creates a new one. In that new graph, each class name ends in +, indicating that the class is now an aggregation of elements rather than a single element. The new graph is called CLASS+. Finally, the `show` command displays the new graph in the Aggregator window.

The command script used for the Duke's Bank system executed the following abstractions that were later used to generate the decomposition style view:

- collapsing the local_variables inside each function to produce the FUNCTION+ graph. Local_variables inside functions are not architecturally relevant.
- collapsing the functions and global variables defined within each file to produce the FILE+ graph
- collapsing the files in which a class is defined inside each class to produce the CLASS+ graph
- collapsing the classes inside each package to produce the PACKAGE graph

Figure 3 shows the result of running the entire script: the PACKAGE graph. This graph shows the decomposition style view of the Duke's Bank system's architecture at the package level. Within the Aggregator window, you can click on the various tabs to see other graphs created by the script.

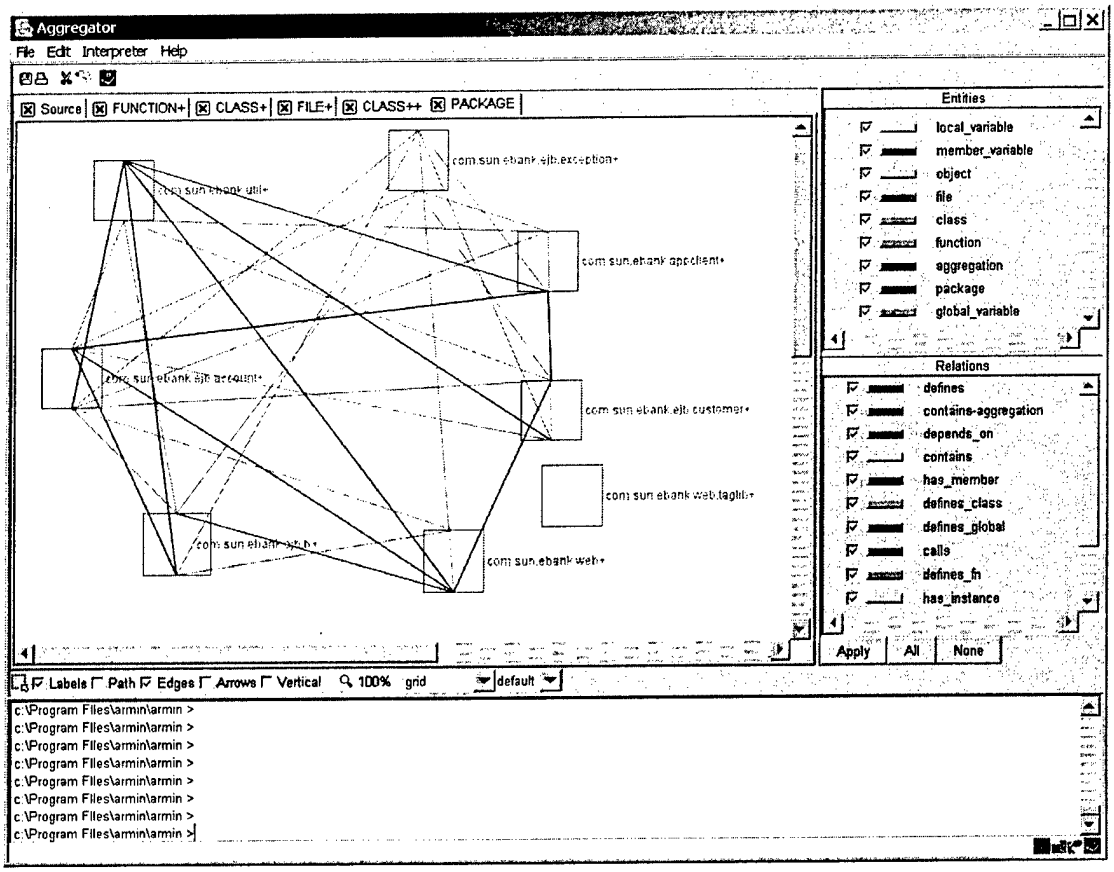


Figure 3: View Showing the Packages and Relationships Produced by ARMIN

Each node in the graph above represents a package. ARMIN lets you view only the package you select and those that are dependent on it. Figure 4 shows an example of this—the "depends_on" relation from Table 1 that represents the importing of a set of classes and their associated methods from one file in one package into a different file in a different package.

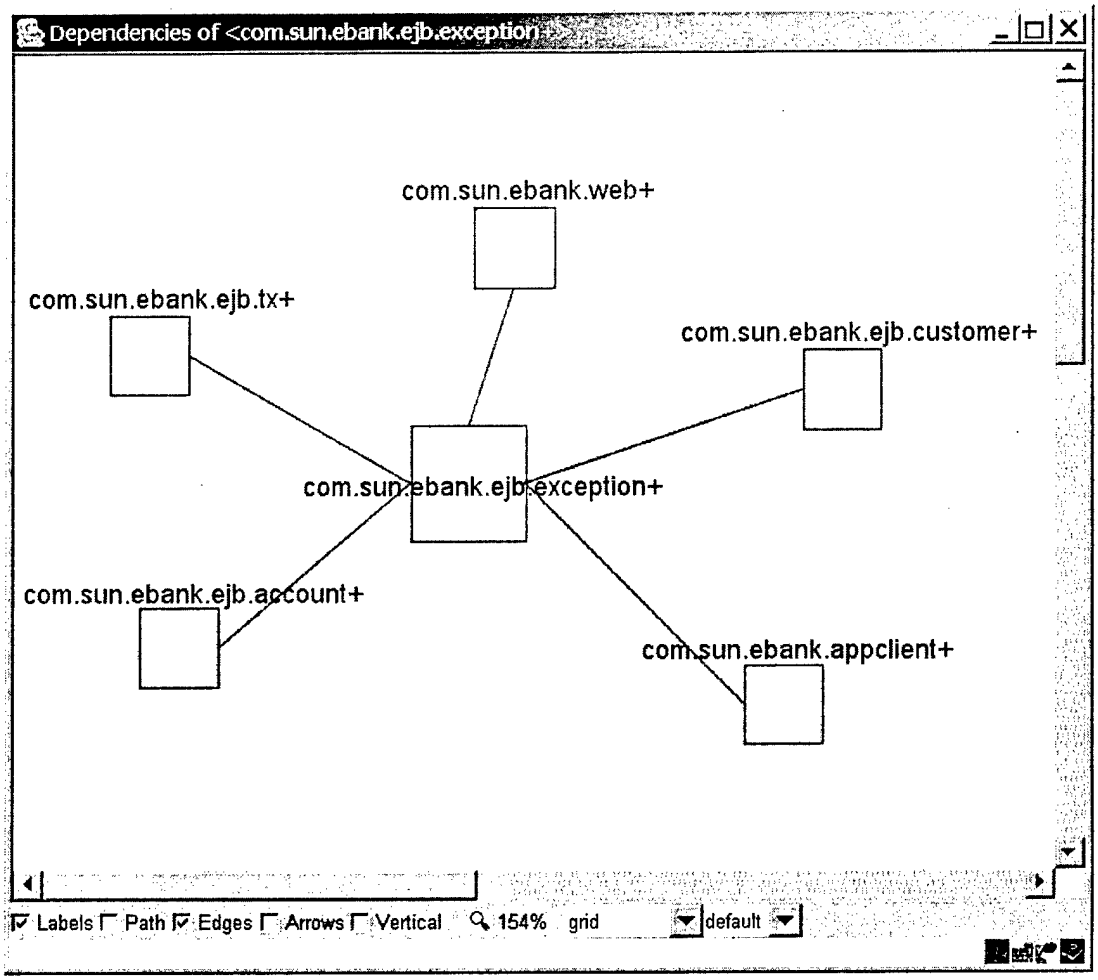


Figure 4: Dependencies for the com.sun.ebank.ejb.exception Package

ARMIN also provides the capability to "drill down" into multiple levels of detail for a particular package. The tool can show the decomposition of the package into classes and further decompose a class into detailed source elements such as files, functions, and variables. The "drill down" views of the system represent decomposition style views of the application. The subgraph shown in Figure 5 contains only the classes that are part of the com.sun.ebank.ejb.exception package.

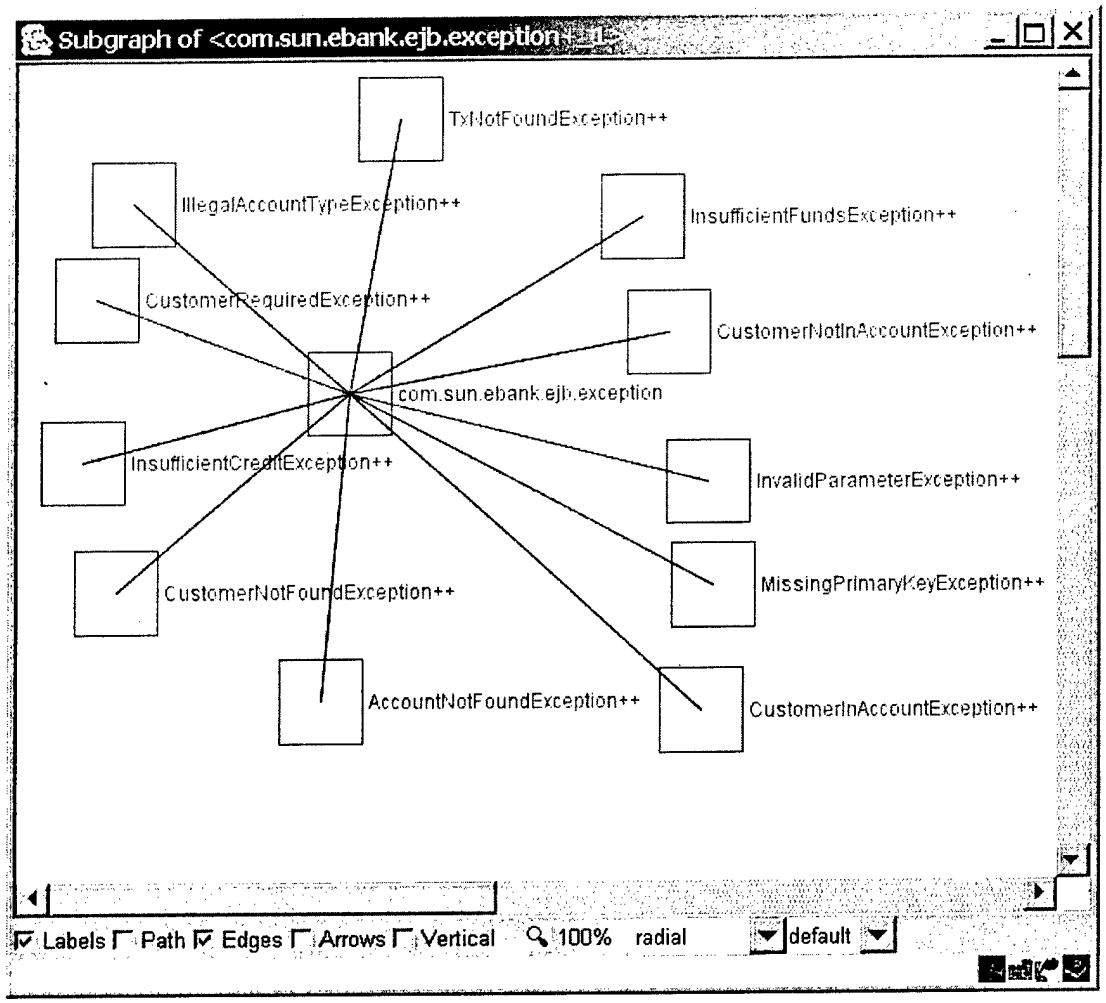


Figure 5: "Drill Down" to the Decomposition View of a Package

So far, we have shown the package decomposition style view for the application and can view the dependencies among them. We can also generate a further abstraction of the source information by identifying other high-level elements of the system. The Duke's Bank system has components such as Customer and Account, which we identified by examining the application's code and documentation. By identifying the set of high-level components and grouping the classes that are part of the representation of each component within the system, we can generate a view of the architecture showing those components and the dependencies among them.

We developed a command script in ARMIN that identified the set of classes that comprise each of the high-level components. An example script for the Customer component is as follows:

```
# create Customer component
$cust={{{"Customer"},{list("Customer*", system.types.class)}}};
$comps.append( $cust );
```

In the above script, a list of all classes beginning with the word "Customer" such as Customer, CustomerBean, CustomerHome, and so forth is created. Each class is then collapsed within the high-level components in which it belongs, and a new view is generated in the Aggregator. Figure 6 shows the view containing these components and the dependencies among them. Again, it is possible to "drill down" to various levels of detail for these components.

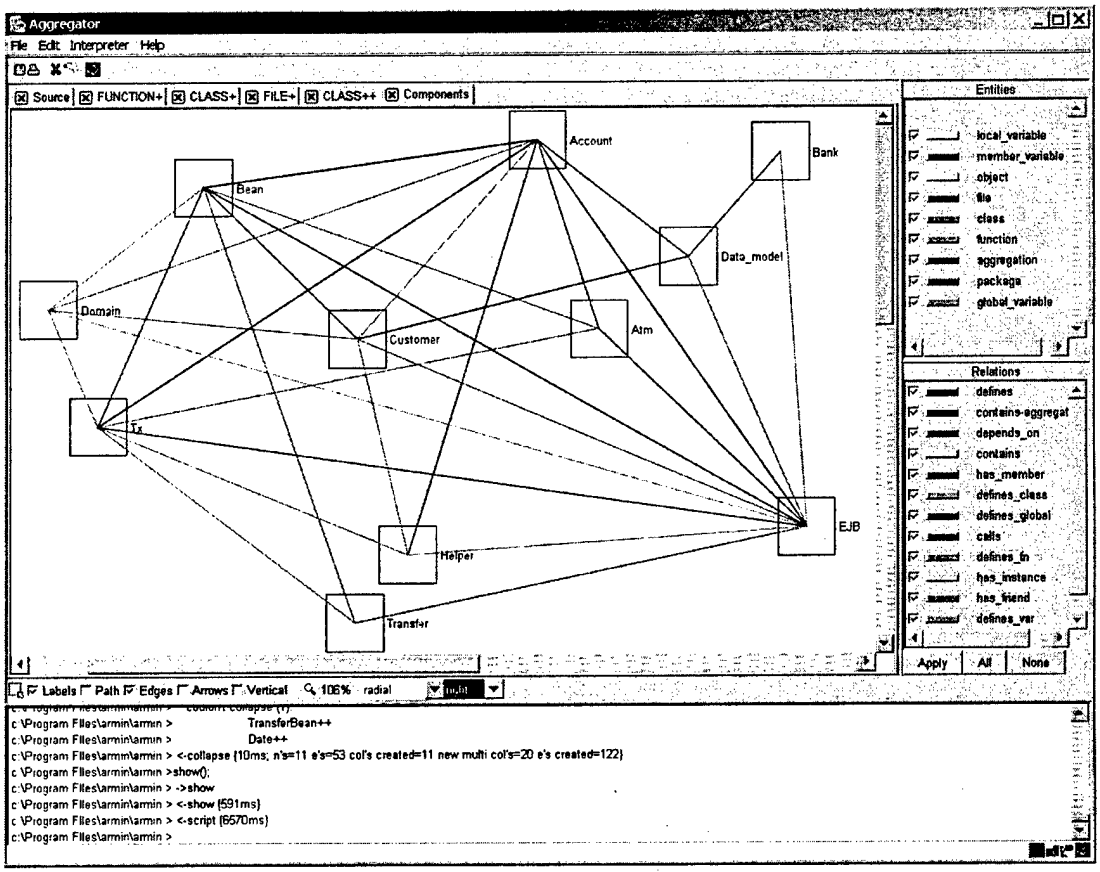


Figure 6: Components and Their Dependencies

By selecting an edge between two components, such as the edge between Account and Data_Model, we can show the details of the dependencies between them. Examples of such dependencies are shown in Figure 7. They include function calls between the components and import dependencies (depends_on relation).

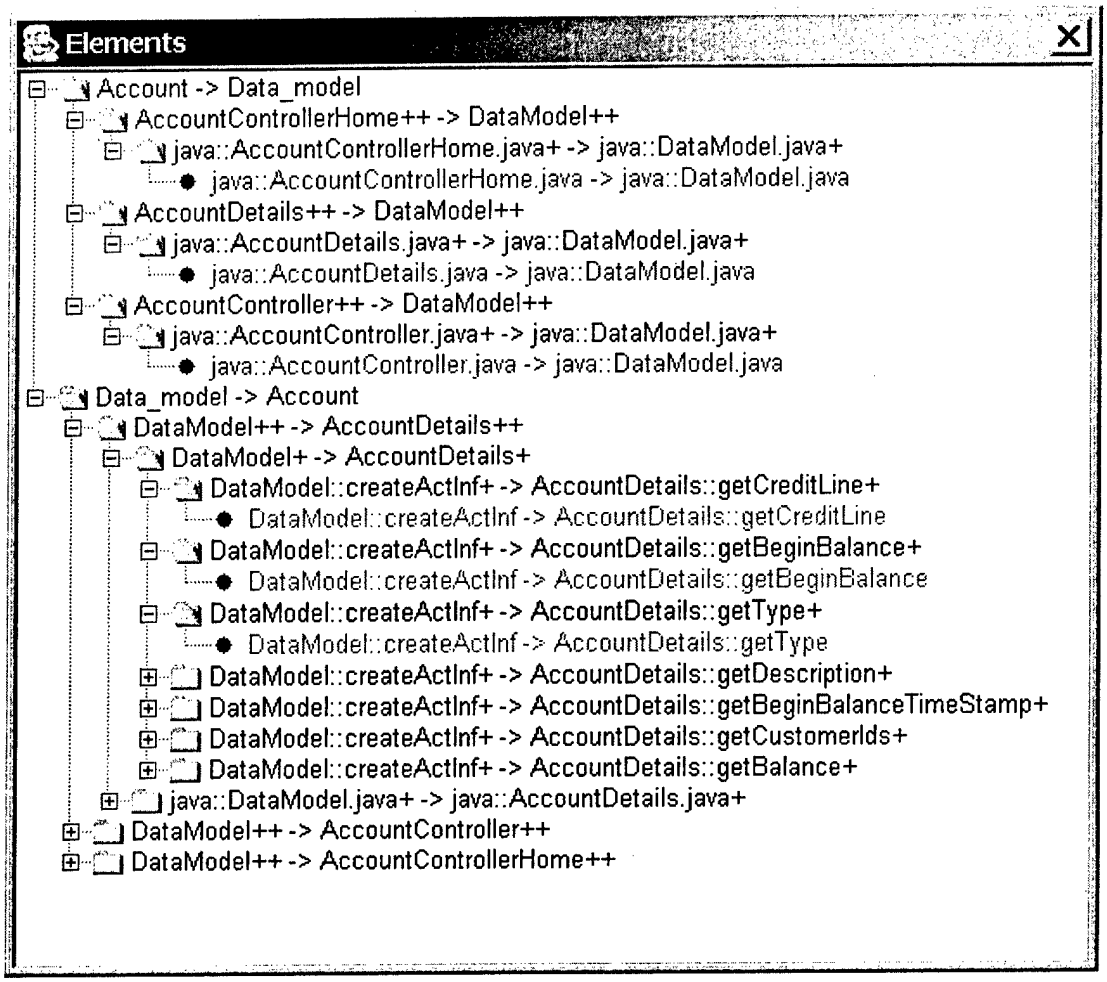


Figure 7: Dependencies Between the Account and Data_Model Components

To verify the decomposition view of the architecture, the directory structure of the source code is analyzed and compared. For example, the containment structure of the class inside the package as shown in Figure 5 can be verified by opening the files inside each Java package to see if the classes in the package match the list produced by ARMIN. Analysis of the decomposition view verified that it contains the right information about the static structure.

We verified the component view of the architecture (shown in Figure 6) by randomly selecting a small set of components and investigating (from the source-code level) the dependencies among them.

These static views of the system show the dependencies between packages and components. The views could be used to help maintain the system, because they highlight the dependencies between the packages and components. It would be difficult to identify those dependencies just by scanning the application's code. These views could support the refactoring of the system if some of the dependencies need to change. Knowing the dependencies involved helps to decide whether a particular component can be reused in a new system.

4 Conclusions and Possible Future Work

Architecture reconstruction can be used to produce different decomposition style views of a system's architecture that are useful for maintaining and documenting the static behavior of a system. Clearly defining the details of the views to be generated before starting the reconstruction process helps to scope the reconstruction work. It also helps to identify which information should be extracted from the source code and later used in the reconstruction process. The set of elements and relation types must be produced before the data containing their instances is extracted from the source code. Such documentation might require knowledge about the different architectural documentation styles and viewtypes, since they represent different elements and relations.

Extracting the instances of the elements and relations from the Understand for Java tool required the most effort. The integrity of the data, extracted from the source code and imported into ARMIN, depends mostly on two tools: Understand for Java and Perl scripts. Tools that can read and parse the source code, and produce text output files should be evaluated to compare the quality and amount of data they produce at both the element and relation level. The tool chosen for this purpose must be able to produce the correct output and be manipulated with Perl scripts so the correct set of element and relation instances can be captured. Although we found the Understand for Java tool to be useful overall in this particular reconstruction study, several of the report files it produced were not. The Perl scripts must be developed and tested carefully during an iterative process.

Creating the ARMIN Interpreter command scripts for abstracting and composing the views of the architecture requires an understanding of the syntax and semantics of the ARMIN Reconstruction Language. Verification of the architecture's decomposition style views can be done with little knowledge of a J2EE/EJB application. However, if the component-and-connector viewtype (which shows dynamic views of the system) is the main focus, a J2EE expert might be needed to do the verification.

The time it takes to extract the source information and produce architectural views using ARMIN is expected to be significantly faster than a manual architecture reconstruction approach. The main reason for this is that the time it takes to go through each package, file, class, function, and variable can be considerable, especially with larger applications.

Possible future work could include generating dynamic views of the Duke's Bank system and generating architectural styles that are part of the component-and-connector viewtype. We will continue to use ARMIN on other reconstruction projects.

References

URLs valid as of the publication date of this document

[Bodoff 03] Bodoff, S.; Green, D.; Jendrock, E.; & Pawlan, M. *The Duke's Bank Application.* <http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html> (2003).

[Clements 03] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond.* Boston, MA: Addison-Wesley, 2003.

[Müller 93] Müller, H. A.; Mehmet, O. A.; Tilley, S. R.; & Uhl, J. S. "A Reverse Engineering Approach to System Identification." *Journal of Software Maintenance: Research and Practice 5*, 4 (December 1993): 181-204.

[O'Brien 01] O'Brien, L. *Architecture Reconstruction to Support a Product Line Effort: Case Study* (CMU/SEI-2001-TN-015, ADA395167). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. <http://www.sei.cmu.edu/publications/documents/01.reports/01tn015.html>.

[O'Brien 03] O'Brien, L. & Stoermer, C. *Architecture Reconstruction Case Study* (CMU/SEI-2003-TN-008). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tn008.html>.

[STI 03] Scientific Toolworks Inc. <http://www.scitools.com/> (2003).

[Sun 03] Sun Microsystems. <http://www.sun.com/index.xml> (2003).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	November 2003	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Architecture Reconstruction of J2EE Applications: Generating Views from the Module Viewtype	F19628-00-C-0003

6. AUTHOR(S)

Liam O'Brien and Vorachat Tamarree

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2003-TN-028

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report outlines the application of architecture reconstruction techniques to the Sun Microsystems' Duke's Bank system—a Java2 Platform, Enterprise Edition/Enterprise JavaBeans (J2EE/EJB) application implemented mainly in Java. The goal of the reconstruction was to apply architecture reconstruction techniques to a system implemented in Java to produce a set of views that depict that system's architecture. Decomposition style views of the module viewtype were used. They focus on the "is part of" relation and show how the system is decomposed into modules and submodules.

During the reconstruction, several decomposition style views of the architecture were generated using the Understand for Java tool. That tool extracted and then abstracted low-level source information from the system. Then that information was formatted using Perl scripts, so it could be loaded into the Architecture Reconstruction and Mining (ARMIN) tool developed by the Carnegie Mellon® Software Engineering Institute and the Robert Bosch Corporation. The resulting views showed the architectural elements of the Duke's Bank system and the dependencies among them.

14. SUBJECT TERMS	15. NUMBER OF PAGES
architecture reconstruction, J2EE/EJB, decomposition style	28

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL